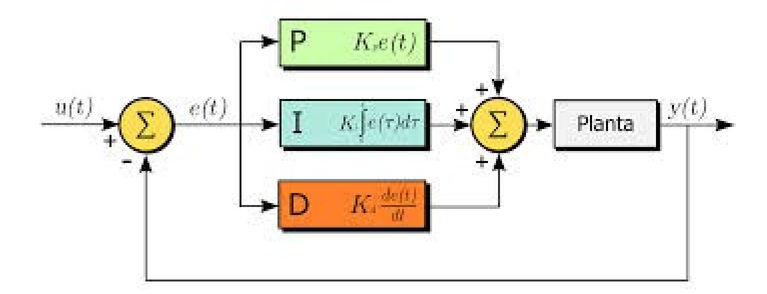
Implémenter un PID sans calcul

Table des matières

Implémenter un PID est-il impossible sans avoir une base solide en mathématiques ? Non, outre l'approche mathématiques, le PID peut très bien s'expliquer de façon intuitive. Le but de cet article est d'expliquer comment marche une régulation PID sans entrer dans les calculs ni sans utiliser les fonctions de transfert du système et autres équations.



1. Approche pragmatique

Imaginez que vous conduisez votre voiture sur l'autoroute et que vous souhaitez stabiliser votre vitesse à 130 km/h exactement ! Dans la pratique, vous y arriverez sans trop de problèmes. Le but de cet exemple est d'essayer de comprendre comment votre esprit arrive à réguler votre vitesse intuitivement.

Règle 1 :

Tout d'abord, la chose la plus intuitive que vous faites lorsque vous voulez rouler à une certaine vitesse sur l'autoroute, c'est de vous dire : " plus je roule lentement par rapport à la vitesse voulu et plus j'appuie sur la pédale d'accélération ". La pression sur l'accélérateur est donc proportionnelle à l'erreur que vous commettez, c'est-à-dire, proportionnelle à la différence entre la vitesse voulue et la vitesse réelle.

Avec cette méthode, pas de problème. Vous êtes à 50 km/h, vous appuyez sur le champignon ! Votre vitesse augmente et se rapproche de la vitesse limite. Vous exercez une pression de moins en moins forte. Une fois que la vitesse atteint 130 km/h, vous avez réussi ! L'erreur est nulle, vous lâchez donc l'accélérateur. Manque de bol, comme l'accélérateur est complètement lâche, la voiture commence à ralentir. Vous recommencez donc à appuyer un peu sur l'accélérateur, puis de plus en plus au fur et à mesure que la voiture ralenti. Au final, vous arrivez à stabiliser votre vitesse à une vitesse inférieure à celle que vous avez choisi, disons 120km/h.

Règle 2 :

C'est alors que vous vous dites : " Zut ! Je n'arrive pas à atteindre la vitesse voulue, il faut que je rajoute une règle supplémentaire à mon raisonnement ! ". Du coup, vous décidez que si votre vitesse reste longtemps sous l'objectif, vous accélérez de plus en plus fort. Vous décidez donc qu'en plus d'accélérer proportionnellement à l'erreur commise, vous allez aussi mémoriser cette erreur au cours du temps. Plus l'erreur globale est importante et plus vous accélérez.

Ainsi, lorsque vous stabilisez votre vitesse à 120km/h, l'erreur globale augmente et vous vous mettez à appuyer de plus en plus fort sur l'accélérateur jusqu'à atteindre 130km/h... et le dépasser ! En effet, arrivé à 130km/h, l'erreur globale est positive, donc vous continuez à appuyer sur l'accélérateur. Arrivé au-delà de 130km/h, l'erreur est négative et fait donc diminuer d'erreur globale. Vous levez donc le pied de l'accélérateur de plus en plus fortement jusqu'à retourner à 130 km/h. Arrivé à 130km/h, rebelote, l'erreur est passé en négatif et vous continuez à décélérer... ainsi de suite jusqu'à finalement arriver à vous stabiliser à 130km/h après de multiples oscillations.

Règle 3 :

Arrivé à 130km/h, vous vous dites : " ça y est, j'y suis ! Mais je n'ai pas été très efficace... Ne faudrait-il pas rajouter une troisième règle afin d'être plus performant ? ". C'est alors que vous décidez d'anticiper votre vitesse. Plus votre vitesse se rapproche de la vitesse optimale, moins vous accélérez et moins elle se rapproche de la vitesse optimale, plus vous accélérez !

Ainsi, si vous vous rapprochez rapidement des 130 km/h, vous vous empressez de lever le pied afin de ne pas dépasser les 130 trop brutalement. Ainsi, vous réduisez les oscillations et vous vous stabilisez rapidement à la vitesse souhaitez !

2. Implémentation d'un PID sur un robot

Un asservissement PID consiste à mémoriser l'erreur, la somme des erreurs et la différence de

l'erreur courante avec l'erreur précédente.

2.1. Le régulateur proportionnel (P : première règle)

La commande de ce régulateur est proportionnelle à l'erreur.

commande = Kp * erreur

Kp est le coefficient de proportionnalité de l'erreur à régler de façon manuelle.

2.2. Le régulateur proportionnel intégral (PI : première et seconde règle)

La commande de ce régulateur est proportionnelle à l'erreur, mais aussi proportionnelle à l'intégrale de l'erreur. On rajoute donc à la commande généré par le régulateur proportionnel, la somme des erreurs commises au cours du temps.

commande = Kp * erreur + Ki * somme_erreurs

Ki est le coefficient de proportionnalité de la somme des erreurs. Il faut aussi le régler de façon manuelle.

2.3. Le régulateur proportionnel dérivé (PD : première et troisième règle)

La commande de ce régulateur est proportionnelle à l'erreur, mais aussi proportionnelle à la dérivée de l'erreur. La dérivée de l'erreur correspond à la variation de l'erreur d'un échantillon à l'autre et se calcule simplement en faisant la différence entre l'erreur courante et l'erreur précédente (c'est une approximation linéaire et locale de la dérivée).

commande = Kp * erreur + Kd * (erreur - erreur_précédente)

Kd est le coefficient de proportionnalité de la variation de l'erreur. Il faut régler ce coefficient manuellement.

2.4. Le régulateur proportionnel intégrale dérivé (PID : première, seconde et troisième règle)

Ici, la commande est à la fois proportionnelle à l'erreur, proportionnelle à la somme des erreurs et proportionnelle à la variation de l'erreur.

commande = Kp * erreur + Ki * somme_erreurs + Kd * (erreur - erreur_précédente)

Vous devez donc faire une mesure sur votre système pour pouvoir calculer l'erreur et ainsi appliquer le PID. Cette mesure est à faire régulièrement à une certaine fréquence d'échantillonnage.

Tous les x millisecondes, faire :

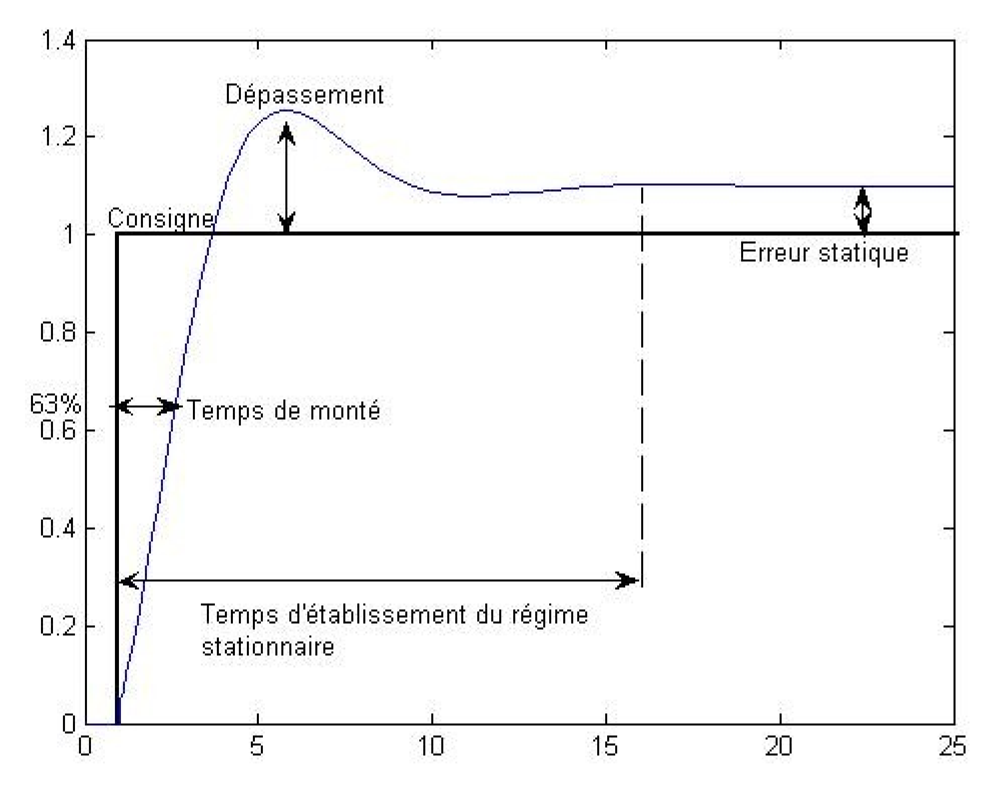
```
erreur = consigne - mesure;
somme_erreurs += erreur;
variation_erreur = erreur - erreur_précédente;
commande = Kp * erreur + Ki * somme_erreurs + Kd * variation_erreur;
erreur_précédente = erreur
```

3. Régler les coefficients d'un PID

Le réglage des coefficients Kp, Ki et Kd d'un PID peut se faire expérimentalement par essais/erreurs. Tout d'abord, sachez qu'il ne sert à rien de vouloir régler les trois coefficients en même temps ! Il y a trop combinaisons possibles et trouver un triplet performant relèverait de l'exploit. Il vaut mieux y aller par étape.

- Tout d'abord, il faut mettre en place un simple régulateur proportionnel (les coefficients Ki et Kd sont donc nuls). Par essais/erreurs, il faut régler le coefficient Kp afin d'améliorer le temps de réponse du système. C'est-à-dire qu'il faut trouver un Kp qui permette au système de se rapprocher très vite de la consigne tout en faisant attention de garder la stabilité du système : il ne faut pas que le système réponde très vite tout en oscillant beaucoup !
- Une fois ce coefficient réglé, on peut passer au coefficient Ki. Celui-là va permettre d'annuler l'erreur finale du système afin que celui-ci respecte exactement la consigne. Il faut donc régler Ki pour avoir une réponse exacte en peu de temps tout en essayant de minimiser les oscillations apportées par l'intégrateur !
- Enfin, on peut passer au dernier coefficient Kd qui permet de rendre le système plus stable. Son réglage permet donc de diminuer les oscillations.

En général, pour régler ces coefficients, on donne au système une consigne fixe (exemple : pour un moteur : tourne à 3 tours par seconde) et on observe la réponse du système (exemple : l'évolution du nombre de tours par seconde du moteur au cours du temps). Le graphe résultant possède donc cette forme :



Le PID parfait n'existe pas, tout est une question de compromis. Certaines applications autoriseront un dépassement afin d'améliorer le temps de stabilisation, alors que d'autres ne l'autoriseront pas (exemple, contrôler un stylo pour écrire sur une feuille. S'il y a dépassement dans le PID, le stylo traversera la feuille). Tout dépend donc du cahier des charges. Chacun des coefficients à un rôle à

jouer sur la réponse à une consigne :

- L'erreur statique, c'est l'erreur finale une fois que le système est stabilité. Cette erreur doit être nulle. Pour diminuer l'erreur statique, il faut augmenter Kp et Ki.
- Le dépassement, c'est le rapport entre le premier pic et la consigne. Ce dépassement diminue si Kp ou Ki diminuent ou si Kd augmente.
- Le temps de montée correspond au temps qu'il faut pour arriver ou dépasser à la consigne. Le temps de montée diminue si Kp ou Ki augmentent ou si Kd diminue.
- Le temps de stabilisation, c'est le temps qu'il faut pour que le signal commette une erreur inférieure à 5% de la consigne. Ce temps de stabilisation diminue quand Kp et Ki augmentent.

Pour vous donner une petite idée de la valeur des coefficients lors de vos premiers essais, vous pouvez regarder du côté de la méthode Ziegler–Nichols. Cette méthode permet de déterminer Kp, Ki et Kd en fonction de votre cahier des charges.

Attention, les coefficients Ki et Kd dépendent de la fréquence d'échantillonnage du système ! En effet, l'intégrateur fera la somme des erreurs au cours du temps ! Si on échantillonne deux fois plus vite, on sommera deux fois plus d'échantillons. Du coup, le coefficient Ki devra être divisé par 2. A l'inverse, pour le dérivateur, si on double la fréquence d'échantillonnage, il faudra doubler le coefficient Kd afin de garder les mêmes performances du PID. Plus la fréquence d'échantillonnage est élevé et plus le PID sera performant. (En effet, plus on échantillonne souvent et plus l'intégration et la dérivée seront précises).

4. Asservissement en vitesse d'un moteur avec Arduino

4.1. Le moteur et la codeuse

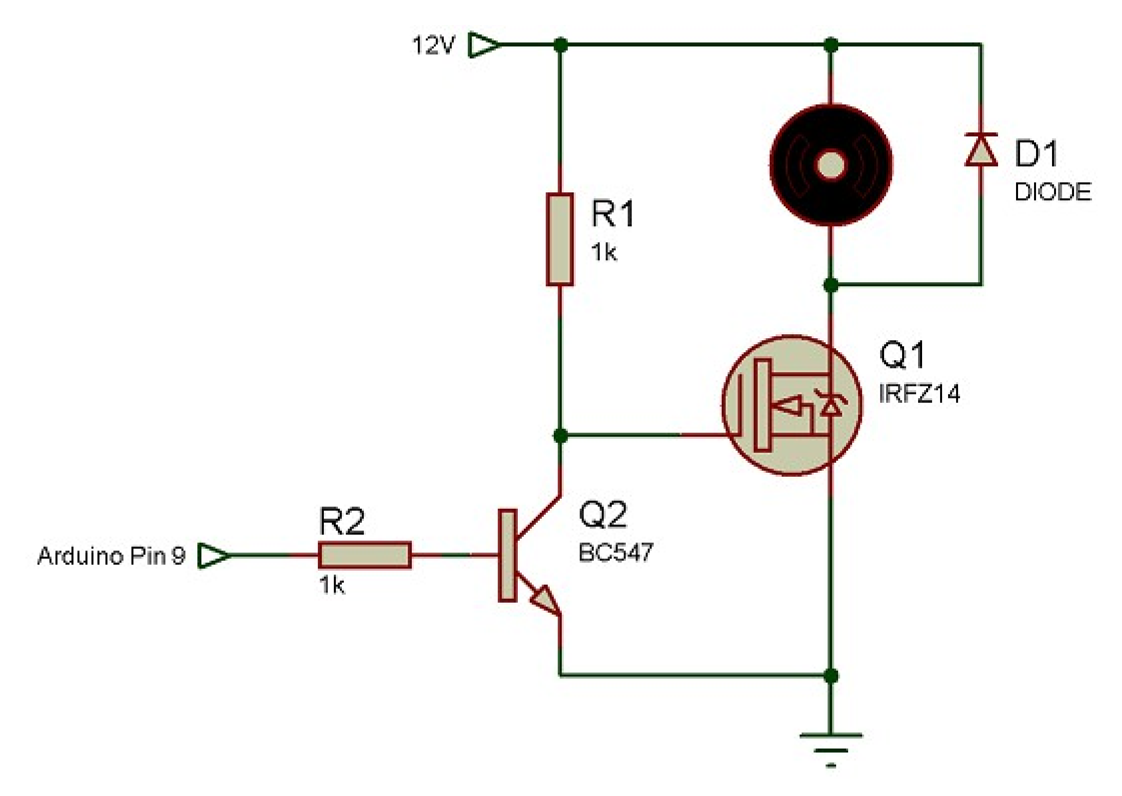
Le moteur que l'on va utiliser est un motoréducteur 29:1 avec une roue codeuse monté sur l'arbre moteur. La documentation est consultable ici.

Les caractéristiques intéressantes à noter sont :

- 350 tours de roue minutes
- 10150 tours d'arbre moteur minute (environ 170 par seconde)
- 32 transitions montante et descendante de la codeuse par tour d'arbre moteur
- 928 transitions montante ou descendante de la codeuse par tour de roue

4.2. Câblage du moteur à l'Arduino

Si on ne dispose pas de shield Arduino pour contrôler un moteur à courant continue, il suffit de câbler une petite interface de puissance très basique.



Le transistor Q2 pilote le MOSFET de puissance. Lorsque la sortie 9 de l'arduino est à l'état haut, Q1 est bloqué et le moteur ne tourne pas. A l'inverse, lorsque la sortie 9 de l'arduino est à l'état bas, alors Q1 devient passant et le moteur tourne à plein régime. Le moteur se contrôle donc en "tout ou rien", ce qui tombe bien, car la sortie "analogique" de l'Arduino est en réalité une sortie PWM de rapport cyclique ajustable.

La sortie 9 de l'Arduino commande le moteur. La codeuse (on n'utilise qu'une seule des deux sorties de la codeuse) est branchée sur la pin 2 qui correspond à l'interruption 0 de l'Arduino (ex : pin n°2 pour une arduino mega).

4.3. Récupération des informations codeuses

Comme nous l'avons vu dans une première partie, il y a 32 transitions montantes ET descendante qui s'opèrent pendant un seul tour de l'arbre moteur.

Ainsi, il suffit de mettre une interruption qui se déclenche à chaque transition de la codeuse et qui exécute une fonction qui viendra simplement incrémenter un compteur.

```
const int _MOTEUR =  9;           // Digital pin pour commande moteur
unsigned int tick_codeuse = 0;    // Compteur de tick de la codeuse

/* Routine d'initialisation */
void setup() {
    pinMode(_MOTEUR, OUTPUT);   // Sortie moteur
    analogWrite(_MOTEUR, 255);  // Sortie moteur à 0
    delay(5000);                // Pause de 5 sec pour laisser le temps au moteur de
s'arréter si celui-ci est en marche

    attachInterrupt(0, compteur, CHANGE);    // Interruption sur tick de la codeuse
(interruption 0 = pin2 arduino mega)
}
```

```
/* Fonction principale */
void loop(){
    delay(10);
}

/* Interruption sur tick de la codeuse */
void compteur(){
    tick_codeuse++;  // On incrémente le nombre de tick de la codeuse
}
```

4.4. Création de la fonction d'asservissement

Pour mettre en place un asservissement numérique, il faut que les calculs de la commande du moteur se fassent à un intervalle de temps régulier. Pour cela, on utilise un timer qui permet d'exécuter une fonction précise tous les x millisecondes. Le timer n'est pas un objets inclut de base à Arduino, il faut installer une bibliothèque externe, SimpleTimer, qui remplit cette tâche. Vous pourrez trouver cette bibliothèque sur cette page.

```
#include <SimpleTimer.h> // http://arduino.cc/playground/Code/SimpleTimer

SimpleTimer timer;                 // Timer pour échantillonnage
const int _MOTEUR =  9;            // Digital pin pour commande moteur
unsigned int tick_codeuse = 0;     // Compteur de tick de la codeuse
int cmd = 0;                       // Commande du moteur
const int frequence_echantillonnage = 50;  // Fréquence d'exécution de l'asservissement

/* Routine d'initialisation */
void setup() {
    Serial.begin(115200);         // Initialisation port COM
    pinMode(_MOTEUR, OUTPUT);   // Sortie moteur
    analogWrite(_MOTEUR, 255);  // Sortie moteur à 0

    delay(5000);                // Pause de 5 sec pour laisser le temps au moteur de
s'arréter si celui-ci est en marche

    attachInterrupt(0, compteur, CHANGE);    // Interruption sur tick de la codeuse
(interruption 0 = pin2 arduino mega)
    timer.setInterval(1000/frequence_echantillonnage, asservissement);  // Interruption
pour calcul du PID et asservissement
}

/* Fonction principale */
void loop(){
    timer.run();
    delay(10);
}

/* Interruption sur tick de la codeuse */
void compteur(){
    tick_codeuse++;  // On incrémente le nombre de tick de la codeuse
}

/* Interruption pour calcul du PID */
void asservissement()
{
```

```
    // Réinitialisation du nombre de tick de la codeuse
    tick_codeuse=0;

    // DEBUG
    Serial.println(tick_codeuse);
}
```

Ici, la fonction asservissement() est exécutée toutes les 20ms (50Hz) et la fonction compteur() est exécuté à chaque fois que la codeuse change d'état.

4.5. Mise en place d'un asservissement P

Essayons maintenant de mettre en place un asservissement proportionnel. Pour cela, il nous faut trois choses.

D'abord, la consigne, qui correspondra, dans notre exemple, au nombre de tours de roue par seconde. On va fixer cette consigne à 5, ce qui veut dire que l'on souhaite que le moteur effectue 5 tours de roue par seconde.

Ensuite, il nous faut le nombre de tour de roue qu'a effectuer le moteur durant les dernière 20ms. Rien de bien compliqué. La variable tick_codeuse compte le nombre de changement d'état de la codeuse durant les 20 dernières millisecondes. On sait qu'il y a 32 changements d'état de la codeuse par tour de l'arbre moteur. On sait qu'il faut 29 tours d'arbre moteur pour faire un tour de roue.

On en déduit donc que la roue à fait tick_codeuse/32/29 tours de roues durant les 20 dernières milliseconde, et donc 50*tick_codeuse/32/29 tours de roue par seconde !

Avec cette information, on peut donc calculer l'erreur qui est la différence entre la consigne (le nombre de tour de roue par seconde voulu) et la réponse à cette consigne (le nombre de tours de roue réalisé par seconde).

Il ne nous reste plus qu'a trouver notre coefficient de proportionnalité de notre régulation, ce qui nous donne la fonction d'asservissement suivante :

```
/* Interruption pour calcul du P */
void asservissement()
{
    // Calcul de l'erreur
    int frequence_codeuse = frequence_echantillonnage*tick_codeuse;
    float nb_tour_par_sec = (float)frequence_codeuse/(float)tick_par_tour_codeuse/
(float)rapport_reducteur;
    float erreur = consigne_moteur_nombre_tours_par_seconde - nb_tour_par_sec;

    // Réinitialisation du nombre de tick de la codeuse
    tick_codeuse=0;

    // P : calcul de la commande
    cmd = kp*erreur;

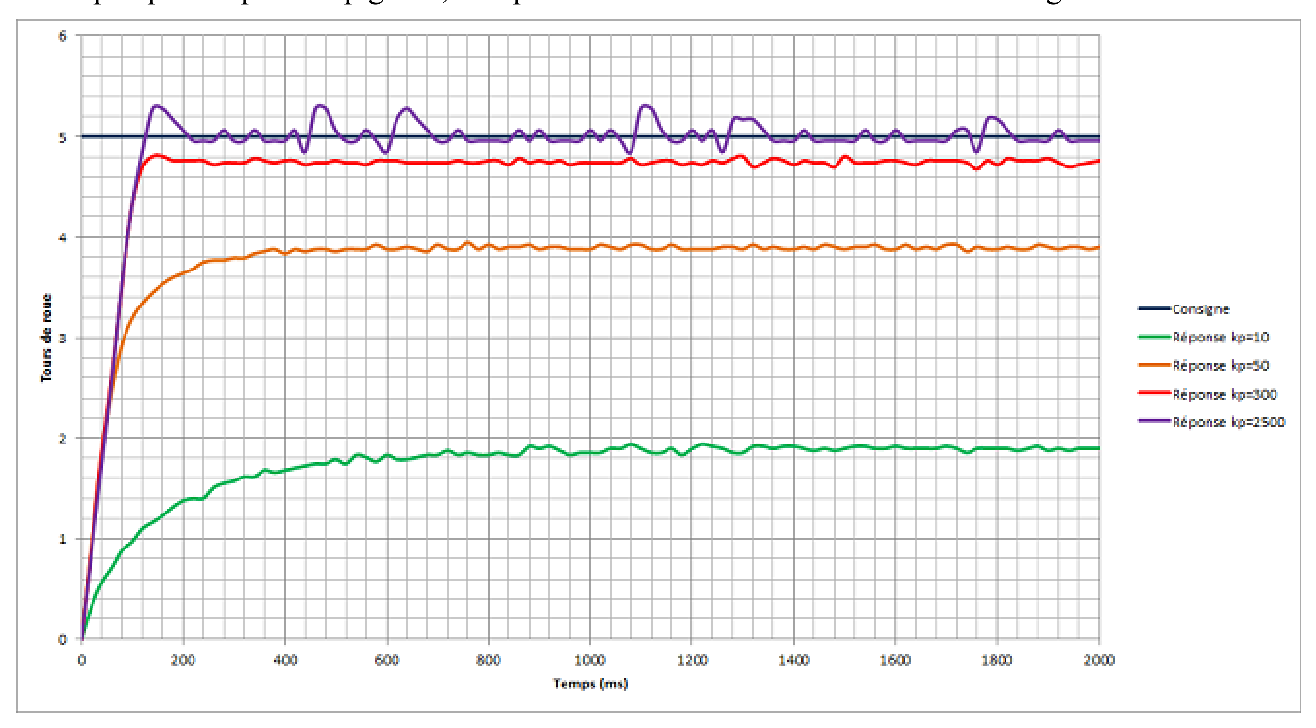
    // Normalisation et contrôle du moteur
    if(cmd < 0) cmd=0;
    else if(cmd > 255) cmd = 255;
    analogWrite(_MOTEUR, 255-cmd);

    // DEBUG
    /*
    Serial.print(nb_tour_par_sec,8);
```

```
    Serial.print(" : ");
    Serial.print(erreur,4);
    Serial.println();
    //*/
}
```

Notons que lors de l'envoie du signal de commande au transistor, il faut inverser le résultat trouvé avec notre asservissement proportionnel car dans notre cas, le moteur tourne pour une commande de 0 et s'arrête pour une commande de 255. D'où la ligne analogWrite(_MOTEUR, 255-cmd).

Il faut donc maintenant définir le coefficient de proportionnalité. On a tracé, ci-dessous, différentes réponses du moteur en fonction du temps pour des coefficients de proportionnalité différents. On remarque bien que quand kp augmente, la réponse se rapproche de plus en plus à la consigne voulu, mais que quand kp est trop grand, la réponse oscille fortement autour de la consigne.



4.6. Amélioration PI

D'après les résultats précédent, on va prendre un coefficient de proportionnalité kp égal à 300. Ainsi, l'erreur statique sera d'environ 5%. Pour améliorer le comportement de notre asservissement et pour annuler notre erreur statique, on a décidé de rajouter un terme intégrateur afin d'obtenir un asservissement PI.

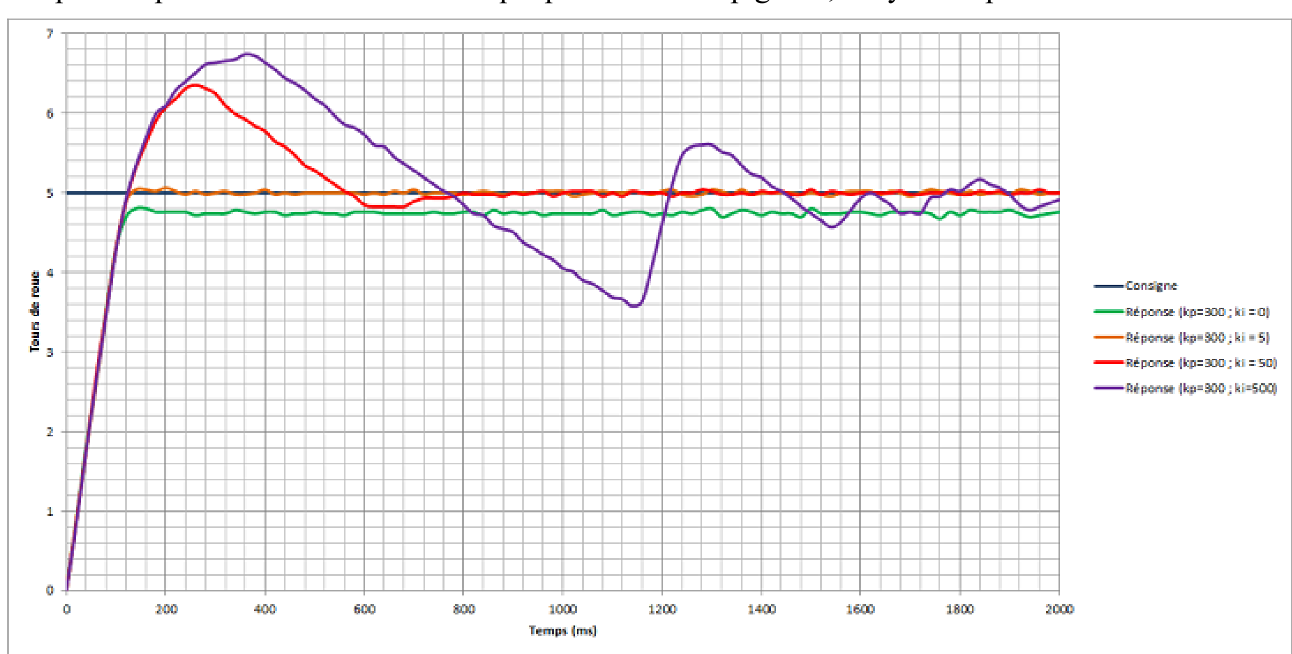
Pour cela, on garde en mémoire la somme de toutes les erreurs du système. Plus cette somme des erreurs est importante et plus on corrige la commande. Ainsi, il faut ajouter ce nouveau terme intégrale à la ligne calculant la commande.

```
somme_erreur += erreur;
// PI : calcul de la commande
cmd = kp*erreur + ki*somme_erreur;
```

Dans cette seconde phase, nous avons donc à régler ce coefficient intégrateur ki.

Plus on augmente le coefficient d'intégration ki, est plus le système répond vite, mais en contre partie, le système devient de plus en plus instable. Le but est donc de trouver un compromis entre

temps de réponse et stabilité. On voit que pour des ki trop grand, le système part en oscillation.

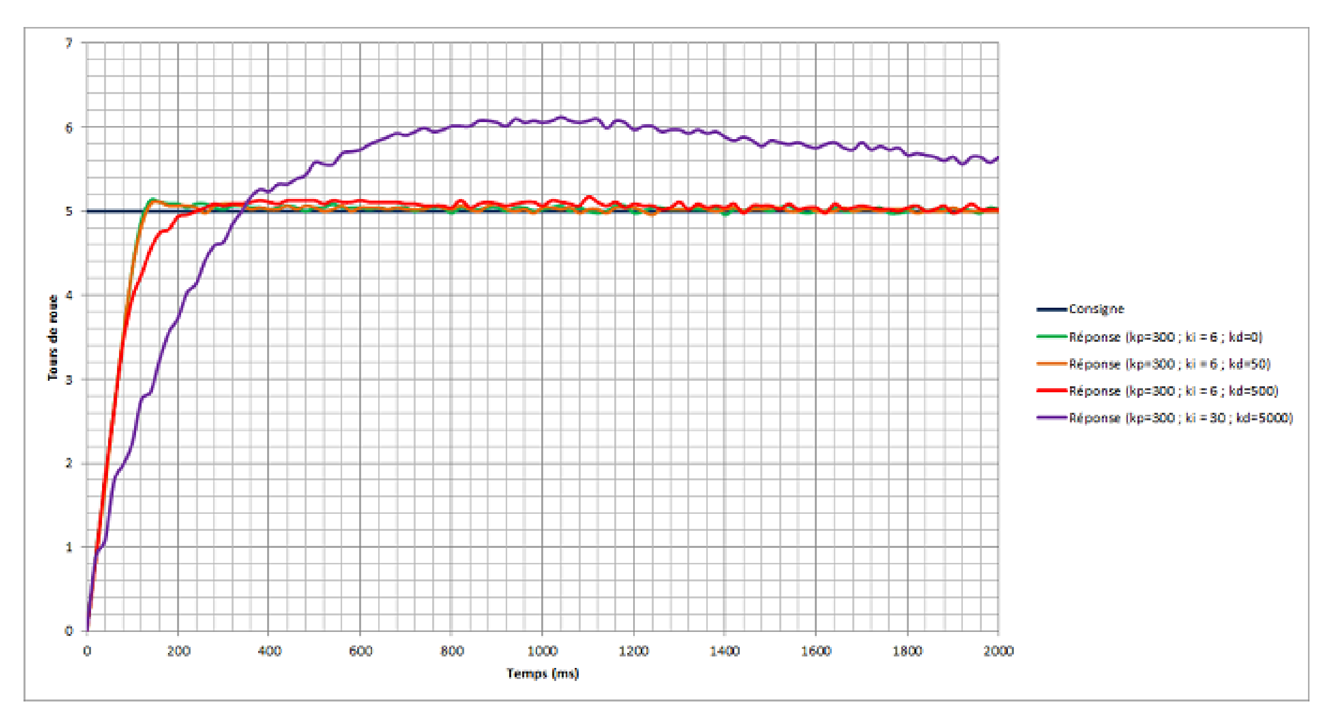


D'après ces graphiques, un coefficient proportionnel de 300 et un coefficient intégrateur de 5 ou 6 nous permet d'atteindre une réponse quasiment optimale, avec un dépassement très faible et un temps de réponse d'environ 120 milliseconde. Le système se stabilise donc après 6 exécutions de la fonction asservissement.

4.7. Asservissement final PID

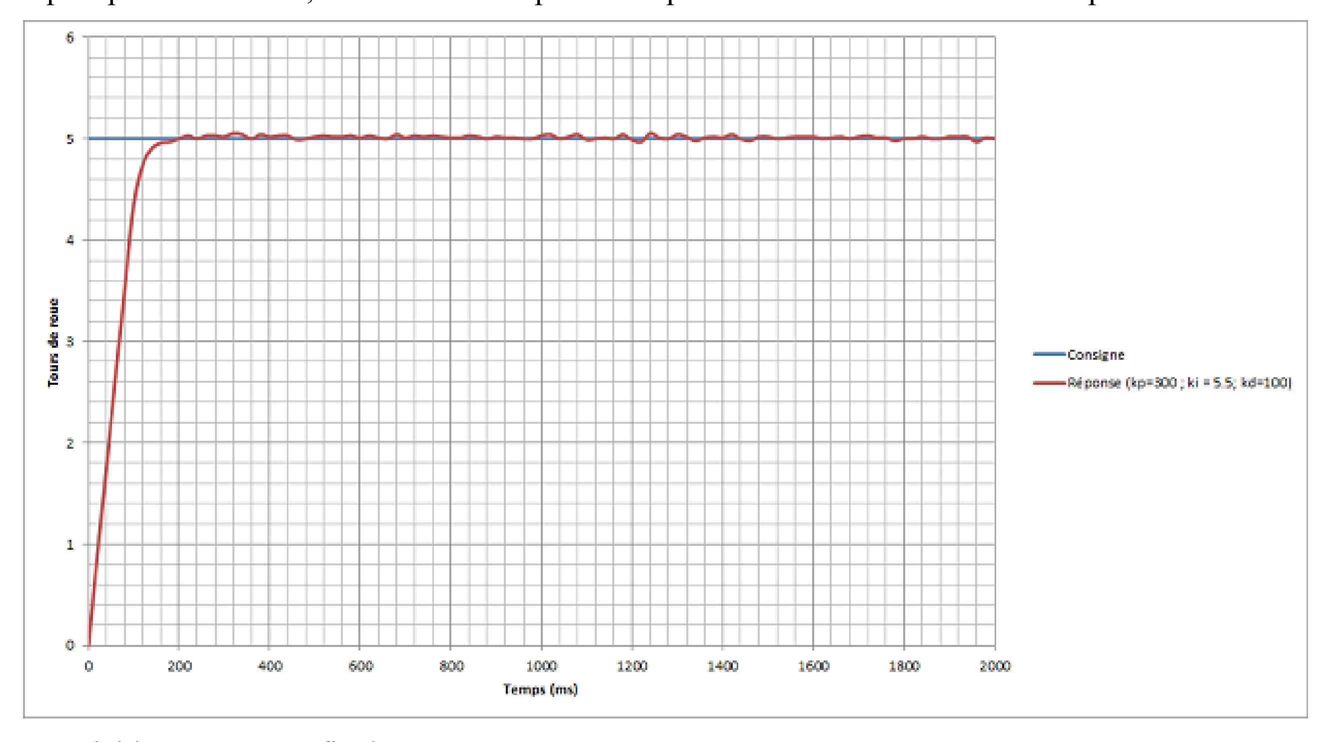
Afin de mettre en place un asservissement PID complet, on va rajouter le terme dérivateur, bien que dans notre cas, celui-ci n'ai pas beaucoup d'influence car un simple asservissement PI nous permet d'atteindre une réponse quasi parfaite.

```
float delta_erreur = erreur-erreur_precedente;
erreur_precedente = erreur;
// PID : calcul de la commande
cmd = kp*erreur + ki*somme_erreur + kd*delta_erreur;
```



Il faut faire attention de ne pas prendre un coefficient dérivateur kd trop grand car le temps de réponse augmente.

Après plusieurs essais, on arrive à un triplet assez performant. Voici l'allure de la réponse :



Et voici le programme final :

```
/**
* Asservissement d'un moteur à l'aide d'un régulateur PID
* Avril 2012 - Black Templar
*/
```

```
#include <SimpleTimer.h>   // http://arduino.cc/playground/Code/SimpleTimer
#define _DEBUG false

SimpleTimer timer;                 // Timer pour échantillonnage
const int _MOTEUR =  9;            // Digital pin pour commande moteur
unsigned int tick_codeuse = 0;     // Compteur de tick de la codeuse
int cmd = 0;                       // Commande du moteur

const int frequence_echantillonnage = 50;  // Fréquence du pid
const int rapport_reducteur = 29;          // Rapport entre le nombre de tours de
l'arbre moteur et de la roue
const int tick_par_tour_codeuse = 32;      // Nombre de tick codeuse par tour de
l'arbre moteur

float consigne_moteur_nombre_tours_par_seconde = 5.;  //  Nombre de tours de roue par
seconde

float erreur_precedente = consigne_moteur_nombre_tours_par_seconde;
float somme_erreur = 0;   // Somme des erreurs pour l'intégrateur
float kp = 300;           // Coefficient proportionnel
float ki = 5.5;           // Coefficient intégrateur
float kd = 100;           // Coefficient dérivateur

/* Routine d'initialisation */
void setup() {
    Serial.begin(115200);         // Initialisation port COM
    pinMode(_MOTEUR, OUTPUT);     // Sortie moteur
    analogWrite(_MOTEUR, 255);    // Sortie moteur à 0

    delay(5000);                  // Pause de 5 sec pour laisser le temps au moteur de
s'arréter si celui-ci est en marche

    attachInterrupt(0, compteur, CHANGE);    // Interruption sur tick de la codeuse
(interruption 0 = pin2 arduino mega)
    timer.setInterval(1000/frequence_echantillonnage, asservissement);  // Interruption
pour calcul du PID et asservissement
}

/* Fonction principale */
void loop(){
    timer.run();
    delay(10);
}

/* Interruption sur tick de la codeuse */
void compteur(){
    tick_codeuse++;  // On incrémente le nombre de tick de la codeuse
}

/* Interruption pour calcul du PID */
void asservissement()
{
    // Réinitialisation du nombre de tick de la codeuse
    int tick = tick_codeuse;
    tick_codeuse=0;

    // Calcul des erreurs
```

```
    int frequence_codeuse = frequence_echantillonnage*tick;
    float nb_tour_par_sec = (float)frequence_codeuse/(float)tick_par_tour_codeuse/
(float)rapport_reducteur;
    float erreur = consigne_moteur_nombre_tours_par_seconde - nb_tour_par_sec;
    somme_erreur += erreur;
    float delta_erreur = erreur-erreur_precedente;
    erreur_precedente = erreur;

    // PID : calcul de la commande
    cmd = kp*erreur + ki*somme_erreur + kd*delta_erreur;

    // Normalisation et contrôle du moteur
    if(cmd < 0) cmd=0;
    else if(cmd > 255) cmd = 255;
    analogWrite(_MOTEUR, 255-cmd);

    // DEBUG
    if(_DEBUG)  Serial.println(nb_tour_par_sec,8);
}
```